

10/560203

Description 1459 Rec'd PCT/PTO 8 DEC 2009

A METHOD, APPARATUS AND COMPUTER PROGRAM FOR PROCESSING A QUEUE OF MESSAGES

Technical Field

[001] The invention relates to data processing systems and more particularly to the processing of a queue of messages by such a system.

[002] It is often necessary to 'replay' a sequence of actions into a database or other system (such as a messaging system).

[003] The sequence may be represented at various levels, e.g. log replay (as used in recovery), transaction change replay (as used in database replication), or call replay (of an audit log of calls made, maybe at the SQL or stored procedure level or even at a higher application level).

[004] Often, execution of the original actions was highly parallelised with database transaction and locking techniques assuring a logical order (sometimes only partial order). This parallelism was essential to achieve good system performance.

[005] The logical order assured by the database transaction and locking techniques is represented in the sequence to be replayed. The problem is that the replay must be as fast as possible, and this also demands some degree of parallelism. However, it is still necessary to preserve the original logical sequence.

[006] By way of example, consider a system where the sequence to be replayed is represented as a queue of work items. Each work item represents an original transaction, and contains a list of database record level updates to be made.

[007] (The term update is used herein to include any operation that changes the database, including at least SQL INSERT, DELETE and UPDATE statements, and also any other updates such as data definition updates.)

[008] The natural (much simplified) implementation of this is a single 'master transaction' thread:

[009] // master transaction thread

[010] for each work item // i.e. originating transaction

[011] get (e.g. read) work item

[012] for each record update in work item

[013] apply update // e.g. record level update

[014] end for each update

[015] commit (work item read operation and database update operation)

[016] end for each work item

[017] However, each 'apply update' (which generally requires the relevant database record to be fetched (e.g. read) before the update can be applied) is likely to need data not currently held in a bufferpool associated with the database, so processing is stalled pending the retrieval of the required data. Thus a problem exists in that processing throughput can be severely impacted.

[018] It will of course be appreciated that there is no corresponding problem with actually getting the update back onto the database disk since standard database lazy write techniques may be used.

[019] Whilst this problem may apply to messaging systems it is less critical. As most queue activity is predictably at the ends of the queues, a bufferpool of messages to be read may be appropriately predictively prefetched. In other words, the queue is typically read in a FIFO order and thus can be predictively and sequentially prefetched.

[020] It is much more difficult to make such predictions in database systems. This is because records sequentially read from a database are typically scattered all over the database disk and work for a database is unlikely to require sequentially (i.e. contiguously) stored data records.

[021] US Patent 6092154 discloses a method of pre-caching data using thread lists in a multimedia environment. A list of data (read requests) which will be required is passed by a host application to a data storage device. In a multimedia environment the data required as a result of a read request can however be easily specified by the host application. In a database environment the host application may not be aware of the way in which the data is stored, or even if it is, the host application may not be able to communicate this appropriately to the data storage device. Thus it is possible that some of the data required by a subsequent operation will not be available.

[022] US Patent 6449696 also discloses prefetching data from disk based on the content of lists with read requests.

Disclosure of Invention

[023] Accordingly the invention provides a method for processing a queue of messages, each message representing at least one request for an update to a database, the method comprising the steps of: browsing a message; extracting from a browsed message an update request; and sending a pretend update request to a database management system (DBMS) responsible for the database which is to be updated, the pretend update request comprising an indication which indicates that the DBMS should not execute the update but should prefetch data that will be required when a corresponding real update is requested.

[024] Preferably the pretend update request is translated into a prefetch request; and required data is prefetched.

[025] Preferably a real update request is subsequently initiated, the real update request using prefetched data in order to execute. Preferably the message comprising the update is destructively got from the queue. Preferably the destructive get is coordinated with the actual database update (two phase commit) so that a copy of the message is not deleted until confirmation of the update(s) has actually been received.

[026] In one embodiment a master thread performs the step of initiating an update request and one or more read ahead thread perform the step of browsing a message.

[027] Preferably the master thread is maintained at a predetermined processing amount behind a read ahead thread. This processing amount could be measured in terms of messages, updates etc. and helps to ensure that data exists in memory when required. If the master thread gets too close there is the danger that required data may not exist in memory when required for an update request to be executed. If on the other hand, the master thread falls too far behind, then there is the danger that memory will become full and result in data that has not yet been used having to be overwritten.

[028] In one embodiment the prefetch request is in a predetermined form which is retained and an identifier is associated therewith in order that the retained predetermined form can be identified and used in subsequent performance of the real update request. Preferably the identifier is returned in response to the pretend update request.

[029] In one embodiment the pretend update is translated into a prefetch request in a predetermined form and associated with an identifier by the DBMS. The identifier is received from the DBMS and is used in issuing a real update request.

[030] Preferably responsive to using prefetched data for an update request, a memory manager is informed that the prefetched data used may be discarded from memory. This helps to avoid memory from becoming over full.

[031] According to one aspect there is provided a method for pre-processing at a database management system (DBMS) update requests to database controlled by the DBMS, the method comprising: receiving an update request at the DBMS; receiving an indication at the DBMS indicating that the update request is a pretend update request and consequently that the DBMS should not execute the update but should prefetch data that will be required when a corresponding real update is requested; translating the pretend update request into a prefetch request; and prefetching required data based on the prefetch request.

[032] Preferably a real update request is subsequently received and already prefetched data is used to execute the real update request.

[033] In one embodiment the prefetch request is in a predetermined form and is retained. An identifier is then preferably associated with the retained predetermined form in order that the retained predetermined form can be identified and used in subsequent

performance of the real update request. Preferably the identifier is returned in response to the pretend update request.

[034] In one embodiment the identifier is received with a real update request and is used in performance of the real update request.

[035] According to another aspect, there is provided an apparatus for processing a queue of messages, each message representing at least one request for an update to a database, the apparatus comprising: means for browsing a message; means for extracting from a browsed message an update request; and means for sending a pretend update request to a database management system (DBMS) responsible for the database which is to be updated, the pretend update request comprising an indication which indicates that the DBMS should not execute the update but should prefetch data that will be required when a corresponding real update is requested.

[036] According to another aspect, there is provided an apparatus for pre-processing at a database management system (DBMS) update requests to database controlled by the DBMS, the apparatus comprising: means for receiving an update request at the DBMS; means for receiving an indication at the DBMS indicating that the update request is a pretend update request and consequently that the DBMS should not execute the update but should prefetch data that will be required when a corresponding real update is requested; means for translating the pretend update request into a prefetch request; and means for prefetching required data based on the prefetch request.

[037] According to another aspect there is provided a computer program for processing a queue of messages, each message representing at least one request for an update to a database, the computer program comprising program code means adapted to perform, when executed on a computer, a method comprising the steps of: browsing a message; extracting from a browsed message an update request; and sending a pretend update request to a database management system (DBMS) responsible for the database which is to be updated, the pretend update request comprising an indication which indicates that the DBMS should not execute the update but should prefetch data that will be required when a corresponding real update is requested.

[038] According to another aspect, there is provided a computer program for pre-processing at a database management system (DBMS) of update requests to database controlled by the DBMS, the computer program comprising program code means adapted to perform, when executed on a computer, the method steps of: receiving an update request at the DBMS; receiving an indication at the DBMS indicating that the update request is a pretend update request and consequently that the DBMS should not execute the update but should prefetch data that will be required when a corresponding real update is requested; translating the pretend update request into a prefetch request; and prefetching required data based on the prefetch request.

Brief Description of the Drawings

- [039] Figure 1 shows a database system in accordance with the prior art;
- [040] Figure 2 illustrates the processing of the present invention in accordance with a preferred embodiment; and
- [041] Figure 3 further illustrates the processing of the present invention in accordance with a preferred embodiment.

Mode for the Invention

- [042] Figure 1 shows a database system in accordance with the prior art. A system 10 has a queue of work items (messages) 20 for processing. Each item of work represents at least one update to a record in database 30.
- [043] Queue of work 20 is read from right to left (FIFO order) and the data required from database 30 can thus be seen as data items (records) A, B, C, D. (Note, each item of work may represent more than one update and may therefore require more than one record from database 30.)
- [044] Apply application 40 reads from queue 20 and requests (via database interface 45) the appropriate data from the Database Management System (DBMS) 70. The DBMS includes a query processor 50 (including a parser) which requests data via bufferpool manager (not shown) from bufferpool 60 (volatile memory). If, as is likely, the data is not found within the bufferpool, then the data must be fetched from the database itself. Because the database may store thousands of records on disk and because the data is unlikely to be stored in a predictable order, system throughput can be severely impacted whilst the required data is retrieved into the bufferpool 60. In this example the queue of work 20 requires records A, B, C and D to be fetched in that order, but the disk stores such records in a completely different order (A, C, D, B). For this reason it is no use for the database to predictively prefetch records (i.e. typically in sequential order) - e.g. after reading A getting C.
- [045] The present invention addresses this problem. Figure 2 illustrates the processing of the present invention in accordance with a preferred embodiment and should be read in conjunction with figure 1.
- [046] A main read ahead thread in the apply application 40 browses each item of work from queue 20 (step 100). As previously discussed, each work item comprises at least one update requested of database 30. For each such requested update a pioneer update thread is spawned (step 110). Each pioneer update thread initiates (via the DBMS 50) the fetching of the data required into bufferpool 60 such that the appropriate update can be applied to the database 30 when requested (step 120).
- [047] The pioneer update threads do not make changes to the database themselves (they only read the data) and thus these may easily be applied in parallel. This parallelism

permits the database to optimize its I/O pattern (in the same way as it would have during parallel execution of the original transactions).

- [048] Full implementation of a pioneer call preferably ensures that all relevant data, both for the record and indices, are read into the bufferpool. For example, take an update that sets the salary of person# 1234 to 50000. This will probably involve the person record itself, and the index to the person table on person#. If there is an index on salary, then that is also preferably prefetched.
- [049] There are three mechanisms by which the pioneer updates (pretend updates) may be achieved:
 - [050] Mechanism A: Where the pioneer thread translates an update request into an associated prefetch request which is a query (i.e. a query to fetch the appropriate data rather than to actually perform the update itself) and issues that request to the database. Thus the pioneer call is simulated with a query on person# 1234.
 - [051] Mechanism B: Where the database interface 45 is extended to permit the pioneer thread to instruct the database that the call is a pioneer call (pretend update request) and not a 'real' update. A pioneer thread extracts an update request from a work item (step 200 of figure 3); sends a the update request to the DBMS (step 210); and informs the DBMS (via an indication with the update request) that this is a pretend update request so that the DBMS can translate the pretend update request into a prefetch request to fetch required data (step 220).
 - [052] Mechanism C: Where the database interface 45 is further extended so that the calls from the pioneer thread (initiating prefetch requests) and from the master transaction thread (i.e. the thread which subsequently performs the requested updates) are explicitly associated; e.g. by passing a token (identifier) for the update between the calls. Note, mechanism C is preferably an extension of mechanism B.
- [053] Mechanism A involves no change to the database. However, it involves more work by the apply application in translating the update into a related query. There is also the possibility that the query will not force the database to read all appropriate data in the bufferpool; in the example the relevant part of index on salary may well not be prefetched. This is because the apply application may not fully understand how the database stores its information or may find it difficult to communicate an appropriate request to the DBMS. These issues are resolved in Mechanism B.
- [054] In mechanism B the pretend update request is translated into a prefetch request and used to fetch data that will be required when a corresponding real update is executed. Because the DBMS creates the prefetch request, it is far more likely that all the required data for a particular real update request will be prefetched. The DBMS is aware of the way in which data is stored and is able to translate this effectively into an appropriate prefetch request.

[055] To explain Mechanism C more fully, when a pioneer update request is spawned by the apply application, it is sent to the query processor in order for the query processor to parse the update request into an internal (predetermined) form. This internal form is used to determine what data to retrieve from the database. Once the data has been retrieved this internal form could be discarded. However in doing this, when the master thread wishes to action the real update, the update request will once again have to be parsed.

[056] To save time and processing power, the query processor can save a parsed internal form resulting from a pioneer update request and this can be associated with a token which is passed back to the apply application. When the master transaction thread wishes to action the update request, the same token can be passed by the apply application to the DBMS and this can then be used to determine the appropriate parsed internal form of that request. This improvement removes the need for double parsing.

[057] The query token of Mechanism C is also preferably used by a bufferpool manager (not shown) to determine when a prefetched data item is not longer needed. When the token is passed from the apply application back to the DBMS in order for an update to be applied, the relevant data is retrieved from the bufferpool and then the token is preferably passed to the bufferpool manager to indicate thereto that the data associated with the token may be removed from the bufferpool. Only in such circumstances is data preferably removed from the bufferpool.

[058] This will reduce the risk of prefetched data being removed from the bufferpool even before it is required, or conversely of being held in the bufferpool too long to the detriment of other data.

[059] It should be noted that whereas Mechanism A could be implemented by the apply application with no changes to the database implementation or interface, Mechanisms B and C require changes to the database interface and implementation.

[060] As alluded to above, the main read ahead thread operates ahead of a master transaction thread (also running in apply application 40). The master transaction thread gets each work item in the same way as the read ahead thread did earlier but this time however the item of work is actually removed from the queue in order to actually action a requested update. Because the read ahead (and pioneer threads) has determined ahead of time the data that will be required by the master transaction thread, the necessary data should already have been fetched into the bufferpool 60. Thus the bufferpool manager should be able to retrieve the data requested by the master thread directly from the bufferpool 60 in order that the requested update can be actioned. Since the requested data is immediately available, I/O is not stalled and thus performance is greatly improved. (As previously stated, standard lazy write techniques can be used to actually get the updated data back onto the disk.)

[061] It is important for reasons of performance that the master transaction thread does not get to close or fall too far behind of the main read ahead thread. (If the master thread falls too far behind then the bufferpool may have to be overwritten with new data when the old data has not yet been used; if the master thread gets too close to the read ahead thread then the data may not have been properly prefetched into the bufferpool before it is required.)

[062] It is thus preferable that there is some form of signalling between the main read-ahead thread and the master transaction thread to prevent this from happening.

[063] Preferably therefore the main read ahead thread (and consequently its pioneer threads) is permitted to get no more than a predetermined amount (**require-dReadahead**) ahead of the master transaction thread (e.g. measured in work items processed, updates processed or bytes processed).

[064] Thus the apply application 40 has two counters, **readaheadAmountProcessed** and **masterAmountProcessed**. In the preferred embodiment, the **requiredReadahead** value is measured in terms of work items processed. Each time a read ahead thread moves to the next work item, **readaheadAmountProcessed** is incremented. Each time processing of a work item is completed by the master thread, **masterAmountProcessed** is incremented. The main read ahead thread includes a sleep loop which causes any pioneer threads controlled by the read ahead thread to also sleep:
while (readaheadAmountProcessed-masterAmountProcessed > requiredReadahead) sleep(10)

[065] In this way, the read ahead thread does not get too far ahead of the main thread. This is important because otherwise there is the danger that the bufferpool 60 will become full thus necessitating the removal of data therefrom which may not have yet been used.

[066] The master thread includes also includes a sleep loop:
while (readaheadAmountProcessed-masterAmountProcessed < requiredReadahead) sleep(10)

[067] This ensures that the master thread does not overtake the main read ahead thread.

[068] Note, the completion of read ahead items may not occur in an orderly manner so it is difficult to precisely define how far the read ahead has reached. Thus this signalling could be made more sophisticated to allow for precisely which pioneer updates are completed.

[069] It should be noted that in certain cases updates are dependent on each other in such a manner that execution of the first update changes the data that must be read in order to implement the later update. For example:

[070] [0] At start, Fred is in Department Y.

[071] [1] Move Fred to Department X.

[074] [2] Change the manager of Fred's Department to Bill

[075] The pioneer execution of [1] will read data for Fred into the bufferpool. This data will later be used when the master transaction thread executes [1]. The pioneer execution of [2] may happen before this update, and will therefore read data for Department Y (Fred's old department) into the bufferpool. However, the real execution of [2] will require data for Department X (Fred's new department) to be in the bufferpool.

[076] In these cases the master transaction thread may stall while executing the corresponding real update (i.e. because the required data is not in the bufferpool). This will impair performance slightly but in general performance will nevertheless be much improved over the prior art methods. Note, it will not cause incorrect results (as out of order processing of the transactions themselves would have done).

[077] Improvements/Variations

[078] Some improvements/variations on the basic idea described above will now be discussed.

[079] The embodiment discussed thus far uses a single read ahead thread with a new thread being spawned (created) for each pioneer update and terminated upon completion of its work.

[080] Thread creation/termination is however an expensive process. Thus a thread pool may be used instead. In this embodiment, a pool of pioneer update threads are continually available and when their work is done they return to the pool for use again at a later time.

[081] Another option (which can be used in conjunction with the thread pool) is to have more than one read ahead thread and for the multiple read ahead threads to share the work. In one embodiment a pool of read ahead threads is also used.

[082] The multiple read ahead threads also preferably signal to each other which work items they are responsible for. In this way one read ahead thread does not try to do work already completed (or in the process of being completed) by another read ahead thread - i.e. effort is not unnecessarily duplicated.

[083] A very simple implementation is one in which the first read ahead thread is responsible for work items 1, 4 and 7; a second read ahead thread browses work items 2, 5 and 8; and a third read ahead thread being responsible for work items 3, 6 and 9.

[084] Another improvement on the basic principle is to use batching. The original transactions execute in parallel so their log forces (for backup purposes) may boxcar (parallel batching). This is useful as log forcing is processor intensive and holds up other processing until the force is complete.

[085] The master transaction thread is single threaded (this is important in order to preserve the logical order) and so boxcaring does not apply. Transaction batching may

however be used with a commit operation being performed only after a certain amount of processing has taken place (measured in terms of time, #messages, #updates, #bytes). Since each commit operation also causes a force operation to the log, transaction batching enables a larger amount of data to be forced in one go rather than the continual interruption to the master transaction thread of multiple (time consuming) log forces.

[086] Another option is to use paired master transaction threads. With a single master transaction thread, this thread would send a batch of updates to the DBMS for processing and then send a commit (or finalise command) thereto. The master transaction thread would then have to wait whilst the DBMS forced the update to the database disk. While waiting for control to be returned from the master thread (from commit), it is preferable for another thread to be processing another batch of updates - e.g. log force of each batch is parallelized with processing of the subsequent batch.

[087] It will be appreciated that whilst the present invention has been described in terms of replaying a sequence of actions into a database it is not limited to such. It is applicable to any environment where there is a queue of messages to be processed involving random access to the data.